Writing C Code 15

Different compilers implement the C language differently, even if they conform to the ANSI standard. This chapter explains the unique features of THINK C: how it implements some features of the language and how it adds to the language. For more detailed information, see Chapter 29.

These other chapters in this manual contain more information on THINK C's implementation of the C language:

- Chapter 18, "Calling Toolbox Routines"
- Chapter 19, "Porting Code"
- Chapter 20, "Writing Inline Assembly"
- Chapter 21, "Using the Standard Libraries," especially the section "Using the Online Standard Libraries Reference"
- Chapter 22, "Object-Oriented Programming"
- Chapter 23, "Using Objects in THINK C"

Contents

How THINK C Implements C 299
Identifier length and capitalization 299
How THINK C looks for header files 299
Once-only headers 299
Using register variables. 300

Integer Representation 302
Integer limits 303

Floating-Point Representation 305
Floating-point parameters and limits 307

THINK C Extensions 310
Disabling trigraphs 312
enums of any size 312
Inline assembly 313
pascal keyword 313
C++ style comments. 313
short double type 313

Comments after #else and #endif 313
Inline function definitions 313
Low-memory global definitions 314
MC68881 unary inline functions 314
Function prototypes with (...) 314
Dimensionless arrays 315
void * with function pointers 315
THINK_C predefined symbol 315

Using #pragma Directives 315
The #pragma parameter directive 315
The #pragma nooptimize directive 316

Accessing Option Settings in Your Code 317
Internal compiler options 318
Set Project Type options 318
Type options 319
Language extension options 319
Enumerated type option 320
Pascal string option 321
Array alignment option 321
Object-oriented programming options 322
Type-checking options 322
Generating processor-specific code options 323
Debugging options 324
Global optimizer options 324
Other optimization options 326

How THINK C Implements C

This section describes how THINK C implements certain parts of the C language, including how it assigns variables to registers and how it represents integers and floating-point numbers.

Identifier length and capitalization

In THINK C, every character in an identifier and its case is significant. For external identifiers, the number of significant characters is 255. For all other identifiers, all characters are significant.

How THINK C looks for header files

Table 15-1 shows the rules THINK C uses to find header files.

#include statement	THINK C...
`<filename.h>`	Looks only in the THINK tree
`"filename.h"`	Looks first in the referencing folder, then in the project tree, and finally in the THINK tree

Table 15-1 Header file search rules

The referencing folder is the folder that contains the file that has the `#include` preprocessor directive. For example, if a source file references a header file `MyUtils.h` and that file in turn has the line `#include "MyUtilTypes.h"`, THINK C will look for `MyUtilTypes.h` in the folder that contains `MyUtils.h` first.

Once-only headers

You may want to create a header file that you include in several places but that defines its symbols only once in a file. You use the `#pragma once` directive for this purpose.

If you have the directive:

```
#pragma once
```

in your header file, THINK C includes that header file only once in any source file. If another file tries to include that header file, THINK C knows that the symbols in that file have already been defined, so it doesn't process the file again.

Keep these restrictions in mind when you use `#pragma once`:

- It's case sensitive. For example, if you include a file once with the statement `#include <stdio.h>` and later with the statement `#include <StdIO.h>`, THINK C will include it twice.
- It doesn't distinguish between files included with `<...>` and `"..."`. For example, suppose you have two header files named `xyz.h`: one in the THINK tree and one in the project tree. If you include one with `#include <xyz.h>` and another with `#include "xyz.h"`, THINK C will not include the second file.
- It ignores characters after the first 32. If you include two files with names that start with the same 32 characters, THINK C will not include the second file.

Note

THINK C 4.0 used a similar mechanism. It would not include a header file if the symbol `_H_`*fname* was defined, where *fname* was the name of the file (in lowercase, without the `.h`). In fact, the `#pragma once` directive does exactly the same thing, but it has the advantage that you can rename the header file without changing the directive.

Using register variables

In THINK C, you can decide which variables are placed into registers or let THINK C decide for you. The Automatic Register Assignment option on the Code Optimization page of the **THINK C options** dialog controls what happens. If that option is on, THINK C automatically assigns variables to registers to make your code as small as possible. If that option is off, THINK C puts only those variables declared `register` into registers.

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" and "Other optimization options" later in this chapter.

If you select this option, the Honor 'register' first option appears, as shown in Figure 15-1. Table 15-2 explains the option's effect.

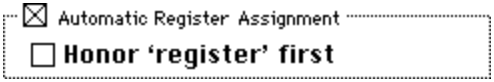

Figure 15-1 Automatic Register Assignment option

If Honor 'register' first is…	THINK C…
On	Puts the variables declared `register` in registers and puts other variables in the remaining registers
Off	Ignores your `register` declarations and places the variables it wants in registers

Table 15-2 Honor 'register' first option

When you ask THINK C to honor your register declarations, the registers in Table 15-3 are available for register allocation.

These registers…	Are used for…
A2-A4	Pointer types
D3-D7	Integral types, pointer types, and floating-point types if Generate 68881 instructions is on
FP4-FP7	12-byte floating-point types if Generate 68881 instructions is on

Table 15-3 Registers available for allocation

Note

If you're building a desk accessory, device driver, or code resource, register A4 is not available for register variables. These project types use A4 to access their globals.

You should not rely on the order in which THINK C allocates variables to registers. THINK C never places a struct or a union in a register, even if it would fit into one.

When you ask THINK C to choose variables to place in registers, THINK C tries to make your code as small as possible. Notice that it doesn't try to save time. For example, it will not make a special effort to place a variable incremented in a loop in a register.

This list explains how to choose the options that are best for your project:

- If you're porting old code with many `register` declarations, turn on Automatic Register Assignment and turn off Honor 'register' first. THINK C will ignore the declarations and will produce the smallest code possible.
- If you're writing new code, turn on both Automatic Register Assignment and Honor 'register' first. As you write, don't use `register` declarations unless you're sure you want to override THINK C's assignments. For example, you may want to put a variable incremented in a loop in a register. Since this option optimizes for space and not speed, THINK C may not put that variable in a register. You must declare it `register` to ensure that it is placed in a register.
- If you need to be absolutely sure what's in each register, turn off Automatic Register Assignment.

Note

When THINK C compiles a function that contains inline assembly, it ignores this option's setting and does not automatically assign variables to registers.

Integer Representation

Integers are represented as two's complement binary numbers. Table 15-4 shows the sizes of integer types.

Type	Bytes
`char`	1
`short`	2
`long`	4
`int`	2 or 4

Table 15-4 Integer sizes

You can test and change the settings of this option from your code. See "Accessing Option Settings in Your Code" and "Type options" later in this chapter.

You choose the size of `int` with the 4-byte int option in the Compiler Settings page of the **THINK C options** dialog. If the option is off, `int` types are 2 bytes long. If it's on, `int` types are 4 bytes long.

Use 4-byte integers if you are doing one of the following:

- Writing a project that mixes C and C++ source files. Symantec C++ does not support 2-byte integers.
- Porting code from a compiler that uses 4-byte integers, like MPW C.
- Porting code that assumes integers and pointers are the same size.

Otherwise, use 2-byte integers, since the MC68000 family handles them more efficiently.

Note

If you turn on the 4-byte int option, you must recompile the ANSI library if your project uses it.

Integer limits

This section describes limits for the THINK C implementation of integers: the largest and smallest values for each integer type. These parameters are defined in `limits.h` as macros. Other implementations of C could have different values.

Changing the values of the macros won't change the way THINK C represents integers. The values are listed in Table 15-7 to indicate the limits of THINK C's integers.

Warning

Don't use the value of a macro in your source code. The value might change in later versions of THINK C. Use the macro's name instead.

Table 15-7 does not list the values for `INT_MIN`, `INT_MAX`, and `UINT_MAX`. The values of these macros depends on the setting of the 4-byte int option on the Compiler Settings page of the **THINK C**

options dialog. If that option is off (the default setting), those macros have the values in Table 15-5.

If 4-byte ints is off, this macro…	Has the same value as this macro…
INT_MIN	SHRT_MIN
INT_MAX	SHRT_MAX
UINT_MAX	USHRT_MAX

Table 15-5 Values for int limits, if 4-byte int option is off

If that option is on, those macros have the values in Table 15-6.

If 4-byte ints is on, this macro…	Has the same value as this macro…
INT_MIN	LONG_MIN
INT_MAX	LONG_MAX
UINT_MAX	ULONG_MAX

Table 15-6 Values for int limits, if 4-byte int option is on

Table 15-7 lists the values for the rest of the macros.

Macro	Value	Description
CHAR_BIT	8	Number of bits for the smallest object that is not a bit-field
MB_LEN_MAX	1	The maximum number of characters that can be in a multi-byte character
SCHAR_MIN	-128	Minimum value for a `signed char`
SCHAR_MAX	127	Maximum value for a `signed char`
UCHAR_MAX	255	Maximum value for an `unsigned` char
CHAR_MIN	-128	Minimum value for a `char`
CHAR_MAX	127	Maximum value for a `char`
SHRT_MIN	-32,768	Minimum value for a `short int`
SHRT_MAX	32,767	Maximum value for a `short int`

Table 15-7 Integer parameters and limits

Macro	Value	Description
USHRT_MAX	65,535	Maximum value for an `unsigned short int`
LONG_MIN	-2,147,483,648	Minimum value for a `long int`
LONG_MAX	2,147,483,647	Maximum value for a `long int`
ULONG_MAX	4,294,967,295	Maximum value for an `unsigned long int`

Table 15-7 Integer parameters and limits *(Continued)*

Floating-Point Representation

THINK C gives you these three types of floating-point numbers:

- `long double` is the fastest floating-point type. Use this type most of the time.
- `float` is smaller than double, but also slower. Use this type when you need to save space and can sacrifice some speed and accuracy.
- `short double` is the medium-sized float but, strangely enough, is the slowest floating-point type. You'll rarely need to use this type.

If you turn on the 8-byte doubles option in a project that uses the ANSI library, you'll need to recompile the ANSI library with that option on. For more information, see the section "Customizing the ANSI library" in Chapter 21.

The THINK C type `double` is usually the same size as a `long double`. However, if the 8-byte doubles option is on, `double` uses the same representation as `short double`. This option is described in "Porting code from other versions of THINK C" in Chapter 19.

Note

The `short double` type is a THINK C extension and is not part of the ANSI standard.

THINK C uses these five different representations for floating-point values:

- 4-byte IEEE single precision
- 8-byte IEEE double precision
- 10-byte SANE extended precision
- 12-byte MC68881 extended precision
- 12-byte universal extended precision

These formats are documented in detail in the Apple Numerics Manual, Second Edition *(Addison-Wesley) by Apple Computer.*

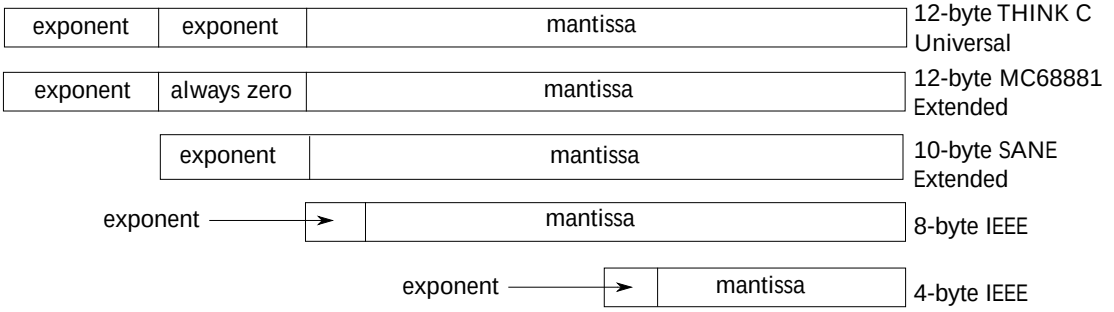

Figure 15-2 Floating-point formats

The universal format is a THINK C extension that lets THINK C use the same format whether the program uses MC68881 code generation or SANE. The universal format is the same as the MC68881 extended format, but it duplicates the exponent part in the second word.

The types `float` and `short double` always use the same representation: `float` uses the 4-byte IEEE representation, and `short double` use the 8-byte IEEE representation. However, the representation for `long double` depends on your option settings.

For more information on the THINK C options dialog, see Chapter 8, "Organizing Projects."

To choose a format for `long double`, use these two options in the Compiler Settings page of the **THINK C options** dialog: Native floating-point format and Generate 68881 instructions. If Native floating-point format is off, THINK C uses the universal format. If Native floating-point format is on, THINK C chooses a format

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code", "Type options", and "Generating processor-specific code options" later in this chapter.

according to the setting of Generate 68881 instructions. Table 15-8 shows which format THINK C uses.

If Native floating-point is...	And Generate 68881 is...	Then doubles use this format...
Off	On or Off	Universal
On	Off	SANE extended
On	On	MC68881 extended

Table 15-8 Choosing a format for long doubles

Note

If your project combines C and C++ source files, or if you're writing a library someone might use with C++, use the native floating-point format. Symantec C++ does not support the universal format.

If you turn on the Native floating-point format option in a project that uses the ANSI library, you'll need to recompile the ANSI library with that option on. For more information, see "Customizing the ANSI library" in Chapter 21.

If you're writing a library that will be used only with C, use the universal format. Projects can use your library no matter how they set the Generate 68881 instructions option, as long as they turn off the Native floating-point format option.

If you want the fastest code possible and use no libraries compiled in the universal format, use the native floating-point format. The conversions to and from the universal format are eliminated.

Floating-point parameters and limits

This section describes some limits and parameters for THINK C's floating-point numbers, such as the largest possible value and the smallest possible fraction. These parameters are defined as macros in the header file `float.h`.

Changing the values of the macros won't change the way THINK C represents floating-point numbers. The values are listed here to let you know the limits and attributes of THINK C's numbers.

Warning

Don't use the value of a macro in your source code. The value might change in later versions of THINK C. Use the macro instead.

For more information on how the Macintosh represents floating-point numbers, see the Apple Numerics Manual, Second Edition *(Addison-Wesley).*

The macros in Table 15-9 define attributes for the THINK C implementation of floating-point numbers.

Macro	Description
`FLT_RADIX`	The radix for exponent representation. THINK C uses 2.
`FLT_ROUNDS`	Specifies how floating-point functions round the results of addition. THINK C rounds to the nearest floating-point value.

Table 15-9 Attributes of THINK C floating-point numbers

The macros in Table 15-11 define limits and parameters for the THINK C implementation of floating-point numbers. The first few letters of the macro indicate the type of numbers for which the macro is used, as in Table 15-10.

Macros starting with…	Describe limits for…
`FLT`	`float`
`DBL`	`double`
`LDBL`	`long double`

Table 15-10 Floating-point macro prefixes

The 8-byte double option on the Compiler Settings page of the **THINK C options** dialog changes the values of the macros beginning with `DBL`. Table 15-11 lists the macro values with that option off (the default setting). Table 15-12 lists the values with that option on. In both charts, all floating-point values are rounded to five decimal places.

Note

The settings of the Generate 68881 instructions and Native floating-point format options on the Compiler Settings page of the **Options** dialog do *not* change the values of these macros.

Macro	Value	Description
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	24 64 64	The number of base FLT_RADIX digits in the floating-point significand
FLT_DIG DBL_DIG LDBL_DIG	6 18 18	Decimal digits of precision
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	-125 -16,382 -16,382	The largest integer that can be a negative exponent when the radix is FLT_RADIX
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	-125 -16,382 -16,382	The largest integer that can be a negative exponent when the radix is FLT_RADIX
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	128 16,384 16,384	The largest integer that can be a positive exponent when the radix is FLT_RADIX
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	128 16,384 16,384	The largest integer that can be a positive exponent when the radix is FLT_RADIX
FLT_MAX DBL_MAX LDBL_MAX	3.40282E+38 1.18973E+4932 1.18973E+4932	The largest representable floating-point number
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209E-7 1.08420E-19 1.08420E-19	The smallest fraction that can be represented in floating-point notation
FLT_MIN DBL_MIN LDBL_MIN	1.17549E-38 1E-4926 1E-4926	The smallest normalized positive floating-point number

Table 15-11 Floating-point parameters and limits

Table 15-12 lists the values for the macros that begin with DBL when the 8-byte doubles option is on.

Macro	Value
DBL_MANT_DIG	53
DBL_DIG	15
DBL_MIN_EXP	-1021
DBL_MIN_10_EXP	-307
DBL_MAX_EXP	1024
DBL_MAX_10_EXP	308
DBL_MAX	1.79769E+308
DBL_EPSILON	2.22045E-16
DBL_MIN	2.22507E-308

Table 15-12 8-byte double parameters and limits

THINK C Extensions

If you're writing ANSI-conformant code, you must disable all the THINK C extensions. For more information, see "Compiling ANSI-Conformant Code" in Chapter 16.

THINK C defines several extensions to the C language that make programming on the Macintosh easier and let you take advantage of object-oriented programming.

Most of the extensions are controlled with the Language Extensions option on the Language Settings page of the **THINK C options** dialog. When you turn that option on, you can choose between two sets of extensions: THINK C or THINK C + Objects.

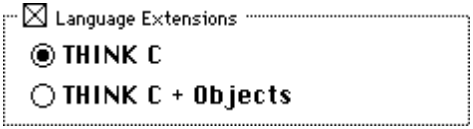

Figure 15-3 The Language Extensions option

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" and "Language extension options" later in this chapter.

The THINK C setting enables these extensions, which are described below:

- Inline assembly
- `pascal` keyword
- C++ style comments
- `short double` type
- Comments after `#else` and `#endif`
- Inline function definitions
- Low-memory global definitions
- MC68881 unary inline functions
- Function prototypes with `(...)`
- Dimensionless arrays
- `void *` with function pointers
- `THINK_C` predefined symbol

The THINK C + Objects setting enables all the extensions above plus it lets you write object-oriented code. The object extensions are described in Chapter 23, "Using Objects in THINK C." If you use the object extensions, these words become keywords and you can't use them as identifiers:

```
class                   delete
new                     virtual
```

How THINK C interprets these words depends on the context. They are not keywords, but you should avoid using them:

```
direct          indirect        inherited
operator        private         protected
public          this
```

Two other extensions have their own option settings on the Language Settings page of the **THINK C options** dialog.

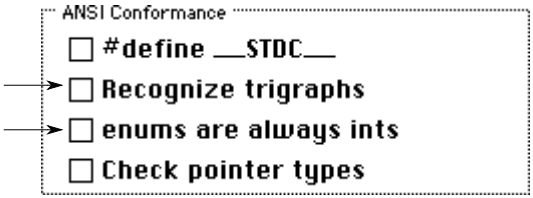

Figure 15-4 Two more THINK C extensions

Table 15-13 shows which extensions these options control.

This extension…	Is controlled by this option…
Disabling trigraphs	Recognize trigraphs
enums of any size	enums are always ints

Table 15-13 THINK C extensions on the Language Settings page

Disabling trigraphs

You can test and change the settings of this option from your code. See "Accessing Option Settings in Your Code" and "Language extension options" later in this chapter.

Trigraphs are sequences of three letters that are treated as one. The sequence contains ?? and an additional character. If you have a computer without such characters as braces ({, }), tildes (~), and carets (^), trigraphs let you use C. However, many Macintosh programs use character literals that resemble trigraphs. For example, the file type `'????'` is interpreted as `'?^`. If you don't want THINK C to recognize trigraphs, turn off the Recognize trigraphs option on the Language Settings page of the **THINK C options** dialog.

Note

To write the file type `'????'` with the Recognize trigraphs option on, use `'???\?'`.

enums of any size

You can test and change the settings of this option from your code. See "Accessing Option Settings in Your Code" and "Enumerated type option" later in this chapter.

In THINK C, enumeration types can be the same size as a `char`, `int`, `short int`, or `long int`.

THINK C makes an enumeration type as small as possible or as large as necessary. For example, this type will only be as large as a `char`:

```
enum { red=1, yellow, green };
```

And this type will be as large as a `long int`:

```
enum { million=1000000,
       billion=1000000000 };
```

If you want enumeration types to be any size, turn off the enums are always ints option on the Language Settings page of the **THINK C options** dialog. If you are writing ANSI-conformant code, turn the option on.

*The other extensions in this section are available when you turn on the Language Extensions option on the Language Settings page of the **THINK C options** dialog.*

Inline assembly

The identifier `asm` is reserved for the inline assembler. For more information about the inline assembler, see Chapter 20, "Writing Inline Assembly."

pascal keyword

The identifier `pascal` is reserved to define functions that follow Pascal calling conventions.

C++ style comments

Two slashes (`//`) introduce a comment. The comment ends at the new-line character.

short double type

The type `short double` is allowed. It is an 8-byte IEEE floating-point number.

Comments after #else and #endif

An identifier after an `#endif` or `#else` preprocessing directive is ignored.

```
#ifdef DEBUG
   printf( "oops!\n");
#endif DEBUG
```

Inline function definitions

You can define inline functions using this form:

```
returnType functionName (arguments) =
   { instr1, instr2, … };
```

THINK C replaces a call to the function *functionName* with the machine instructions *instr1*, *instr2*, … instead of generating a function call. For example, the function `PrOpen()` is defined like this:

```
pascal void PrOpen(void)
   = {0x2F3C,0xC800,0x0000,0xA8FD};
```

When you call `PrOpenPage()`, THINK C generates these instructions:

```
move.l  #0x10000808, -(SP)
_PrGlue
```

For a detailed description of the #pragma parameter directive, see "The #pragma parameter directive" later in this chapter.

You can use the `#pragma parameter` directive to assign registers to parameters:

```
#pragma parameter __A0 NewHandleClear(__D0)
pascal Handle NewHandleClear( Size
   byteCount ) = 0xA322;
```

Most of the Macintosh Toolbox routines are defined in the header files as inline functions.

Low-memory global definitions

Global variables may be defined to refer to absolute memory addresses.

```
extern short MemErr : 0x220;
```

This construct is only legal in global scope. The `extern` storage-class specifier is optional. It was required in earlier versions of THINK C.

MC68881 unary inline functions

When the Generate 68881 instructions option is on, you can declare unary inline functions for the MC68881 like this:

returnType functionName (*argument*) : *instr*;

where *instr* is the low seven bits of the second word of the desired MC68881 instruction. This syntax works with unary functions—functions with only one argument.

For example, this is the inline function definition for `atanh()`:

```
double atanh (double) : 0x0D;
```

The header file `math.h` defines some unary functions in this way. You may want to define others yourself.

Function prototypes with (...)

The parameter list `(...)` is allowed in a function prototype.

Dimensionless arrays

Dimensionless arrays are allowed as the last member of `struct` definitions. The member does not contribute to the size of the `struct`. For example:

```
struct {
   short count;
   char data[];
} CountData;   /* sizeof(CountData) is 2 */
```

void * with function pointers

You can use the type `void *` with function pointers, in addition to data pointers.

THINK_C predefined symbol

THINK C predefines a preprocessor symbol, `THINK_C`, so you can test for THINK C and for the version when doing conditional compilation. In this version of THINK C, it is set to `6`. In THINK C 5.0, it is set to `5`. In THINK C 4.0, it's set to `1`. In previous versions, it's not defined.

Using #pragma Directives

THINK C implements the following `#pragma` directives:

```
once                    parameter
options                 nooptimize
```

The `#pragma options` directive is described in "Accessing Option Settings in Your Code" later in this chapter. The `#pragma once` directive is described in "Once-only headers" earlier in this chapter. The `#pragma parameter` and `#pragma nooptimize` directives are described below.

The #pragma parameter directive

This directive applies to a subsequent inline function definition and allows parameters to be passed in registers instead of on the stack. It specifies which register holds the return value and which registers parameters are passed in. The `#pragma parameter` directive must appear before the inline declaration.

`#pragma parameter` *return-reg*$_{opt}$ *function-name* (*param-reglist*$_{opt}$)

Function-name is the name of a function that is subsequently defined inline. If the definition is not an inline definition, never defined, or already defined, the directive is ignored.

The optional *return-reg* can be `__A0`, `__A1`, `__D0`, or `__D1`.

The optional *param-reglist* is a parameter list made up of `__A0`, `__A1`, `__D0`, `__D1`, or `__D2`.

The inline definition must have a prototype, and the prototype must not end with an ellipsis (`...`). The return type must be an integer type or a pointer type. The return type must be 4 bytes long if the return is in an address register. The argument types may not exceed 4 bytes, except when the inline definition is declared `pascal`, in which case the address of the parameter is used. An address register can hold a 2-byte or 4-byte value for arguments only. In no case can an address register hold a 1-byte value.

Here are some examples of the `#pragma parameter` directive:

```
#pragma parameter __A0 NewHandleClear(__D0)
pascal Handle NewHandleClear( Size
    byteCount )
 = 0xA322;

#pragma parameter Delay(__A0,__A1)
pascal void Delay( long numTicks,
   long *finalTicks )
 = {0xA03B,0x2280};
```

The #pragma nooptimize directive

The `nooptimize` pragma directive allows the appearance of a call to a function to disable global and register optimizations for the function that makes the call. The directive disables register optimizations by turning off automatic register assignment and honoring `register` declarations.

```
#pragma nooptimize(function-name)
```

Function-name must have already been declared as a function; otherwise, it is an error.

Any function that calls a function that has been the subject of the `nooptimize` directive will be compiled without the global and register optimizations, even if those options are selected. For an example, look at the implementation of `setjmp`.

This directive should only be used for functions that have atypical flow-of-control properties.

Accessing Option Settings in Your Code

You can use preprocessor directives to change and test most of the options in the THINK C options and **Set Project Type** dialogs. The `__option` directive lets you query option settings and the `options` pragma directive lets you change them. Both directives take option names as arguments, which are described later in this section.

Note

There are two underscores in `__option`.

To see if an option is enabled, use the `__option` directive in an `#if` directive. For example, this code fragment sets the macro `INT_SIZE` depending on how you set the 4-byte ints option:

```
#if __option(int_4)
   #define INT_SIZE 4
#else
   #define INT_SIZE 2
#endif
```

To change the setting of an option, use the `options` pragma directive. It takes any number of option names, separated by commas. To turn an option on, include its name in the list. To turn an option off, put an `!` in front of its name. For example, this code fragment turns on the Generate 68020 instructions option and turns off the Generate 68881 instructions option:

```
#if MACINTOSH_LC
   #pragma options(mc68020, !mc68881)
#elif
   ...
#endif
```

You can place an `options` pragma anywhere in your file. If it appears outside a function, it applies for the remainder of the file. If it appears inside a function, it applies to that entire function only. The previous settings are restored when THINK C finishes compiling that function. You can change only certain options inside a function.

Note

If an option is set more than once in a function, only the last setting is honored.

Internal compiler options

THINK C sets these options automatically when it compiles your project. You can't change them with the `options` pragma directive, and you can't set them from any dialog. But you can test them with the `__option` directive.

If this is set…	Then THINK C …
`a4_globals`	Uses A4 offsets (instead of A5) for globals. This option is set if your project is not an application.
`pcrel_strings`	Generates strings in code. This option is set if your project is a single-segment code resource.
`jump_table`	Generates jump table. This option is set if your project is an application or a multi-segment desk accessory, device driver, or code resource.

Table 15-14 Internal compiler options

Set Project Type options

These are options in the **Set Project Type** dialog. You can't change them with the `options` pragma directive, but you can test them with the `__option` directive.

If this is set…	Then THINK C…
`far_code`	Generates 32-bit code references. This option is like the Far Code option in the **Set Project Type** dialog.

Table 15-15 Set Project Type options

If this is set…	Then THINK C…
`far_data`	Generates 32-bit data references. This option is like the Far Data option in the **Set Project Type** dialog.
`separate_strs`	Places strings in STRS section. This option is like the Separate STRS option in the **Set Project Type** dialog.

Table 15-15 Set Project Type options *(Continued)*

Type options

These options control how THINK C interprets floating-point and integer declarations. They correspond to options on the Compiler Settings page of the **THINK C options** dialog. You can't change them with the `options` pragma directive, but you can test them with the `__option` directive.

If this is set…	Then…
`int_4`	Integers are 4 bytes long, not 2. This option is like turning on the 4-byte ints option.
`double_8`	Doubles are 8 bytes long. This option is like turning on the 8-byte doubles option.
`native_fp`	Floating-point numbers use the native format, instead of the universal format. This option is like turning on the Native floating-point option.

Table 15-16 Type options

Language extension options

These options control whether you can use some THINK C extensions to the C language. They correspond to options on the Languages Settings page of the **THINK C options** dialog. You can't

change them with the `options` pragma directive, but you can test them with the `__option` directive.

If this is set…	Then THINK C…
`stdc`	Defines the macro `__STDC__`. This option is like turning on the #define __STDC__ option.
`trigraphs`	Recognizes trigraph symbols. This option is like turning on the Recognize trigraphs option.
`thinkc`	Uses the THINK C extensions to the C language.
`objectc`	Uses the object-oriented programming extensions to the C language.

Table 15-17 Language extension options

Depending how `thinkc` and `objectc` are set, you get one of three possible settings for the Language extension level option.

If thinkc is…	And objectc is…	You have this setting…
Off	Off	ANSI C
On	Off	THINK C
On	On	THINK C + Objects

Table 15-18 Language extension level option

The **THINK C options** dialog doesn't let you set `thinkc` off and `objectc` on.

Enumerated type option

This option lets you choose what size enumerated types can be. It corresponds to the enums are always ints option on the Compiler Settings page of the **THINK C options** dialog. You can change it outside a function, but not inside one.

If this is set…	Then…
`pack_enums`	Enumerated types can be the size of any integral type.

Table 15-19 Enumerated type option

Pascal string option

This option lets you choose the type of Pascal string literals. It corresponds to the "\p" is unsigned char [] option on the Compiler Settings page of the **THINK C options** dialog. You can change it outside a function, but not inside one.

If this is set…	Then…
`signed_pstrs`	Pascal string literals (`"\p"`) are of type `char []`.

Table 15-20 Pascal string option

Array alignment option

This option lets you choose whether THINK C aligns even-length `char` arrays within structs. This option corresponds to the Align arrays of char on the Compiler Settings page of the **THINK C options** dialog. You can change it outside a function, but not inside one.

If this is set…	Then…
`align_arrays`	THINK C aligns even-length `char` arrays within structs.

Table 15-21 Array alignment option

For example:

```
struct S {
   char f1;
   char f2[7];
   char f3;
   char f4[8];
   char f5;
};
```

Normally, THINK C allocates the member `f4` on an even boundary and inserts a pad byte between `f3` and `f4`. The entire struct occupies 20 bytes. If the `align_arrays` option is off, THINK C allocates `f4` on an odd boundary and doesn't insert a pad character. The struct occupies only 18 bytes.

Note

THINK C always aligns arrays of integers in structs, regardless of the value of this option.

Object-oriented programming options

These options let you compile object-oriented code written for a previous version of THINK C. The first two correspond to options on the Compiler Settings page of the **THINK C options** dialog. You can change them outside a function, but not inside one.

If this is set…	Then…
`virtual`	Methods are virtual by default. This option has the same effect as turning on the Methods are virtual by default option.
`indirect`	Classes are indirect by default. This option has the same effect as turning on the Classes are indirect by default option.
`class_names`	THINK C generates class names in your application so you can use the functions `new_by_name()` and `class_name()`. If you use very long class names, turning this option off significantly reduces the size of your global data. This option has the same effect as turning on the Generate class names option.

Table 15-22 Object-oriented programming options

Type-checking options

These options let you control how strictly THINK C type-checks your code. They correspond to options on the Languages Settings page of the **THINK C options** dialog. You can change them outside a function, but not inside one.

If this is set…	Then THINK C…
`check_ptrs`	Checks pointer types. This option has the same effect as turning on the Check pointer types option.
`require_protos`	Requires function prototypes.
`infer_protos`	Infers a prototype when a function is first used.

Table 15-23 Type-checking options

With `require_protos` and `infer_protos`, you can get one of the three possible settings for the Strict Prototype Enforcement option.

If require_ protos is…	And infer_ protos is…	You get this setting…
Off	Off	Strict Prototype Enforcement option off
Off	On	Strict Prototype Enforcement option on, and Infer Prototypes option selected
On	On or Off	Strict Prototype Enforcement option on, and Require prototypes option selected

Table 15-24 Strict Prototype Enforcement option

Generating processor-specific code options

These options let you generate code for a specific processor. They correspond to options on the Compiler Settings page of the **THINK C options** dialog. You can change them anywhere.

If this is set…	Then THINK C…
`mc68020`	Generates code for the MC68020. This option has the same effect as turning on the Generate 68020 code option.
`mc68881`	Generates code for the MC68881. This option has the same effect as turning on the Generate 68881 code option.

Table 15-25 Processor-specific code options

Note

If `native_fp` is on, you can't change the setting of the `mc68881` option.

Debugging options

These options help you debug your code. They correspond to options on the Debugging page of the **THINK C options** dialog. You can change them anywhere.

If this is set…	Then THINK C…
`profile`	Generates calls to the profiler. This option is like turning on the Generate profiler calls option.
`force_frame`	Generates a stack frame for each function call. This option has the same effect as turning on the Always generate stack frames option.
`macsbug_names`	Generates MacsBug names.
`long_macsbug_names`	Uses long format for MacsBug names.

Table 15-26 Debugging options

With `macsbug_names` and `long_macsbug_names`, you can get one of the three possible settings for the MacsBug Names option.

If macsbug_ names is…	And long_macs bug_names is…	You get this setting…
Off	On or Off	MacsBug Names option off
On	Off	MacsBug Names option on, and Short format option selected
On	On	MacsBug Names option on, and Long format option selected

Table 15-27 MacsBug Names option

Global optimizer options

These options control the global optimizer, described in “Using the Global Optimizer” in Chapter 17. They correspond to options on the

Debugging page of the **THINK C options** dialog. You can change them anywhere.

If this is set…	Then…
`global_optimizer`	Use the global optimizer on this code. This option is like turning on the Use Global Optimizer option.
`gopt_induction`	If the global optimizer is on, simplify induction expressions. This option is like turning on the Induction variable elimination option.
`gopt_cse`	If the global optimizer is on, eliminate common expressions. This option is like turning on the CSE elimination option.
`gopt_loop`	If the global optimizer is on, move loop invariants. This option is like turning on the Code motion option.
`gopt_coloring`	If the global optimizer is on, optimize register usage. This option is like turning on the Register coloring option.

Table 15-28 Global optimizer options

Note

Inside a function, you can turn off `global_optimizer`, but you cannot turn it on.

Other optimization options

These options control other optimizations. They correspond to options on the Code Optimization page of the **THINK C options** dialog. You can change them anywhere.

If this is set…	Then THINK C…
`defer_adjust`	Defers and combine stack adjusts. This option is like turning on the Defer & combine stack adjusts option.
`redundant_loads`	Suppresses redundant loads. This option is like turning on the Suppress redundant loads option.
`honor_register`	Honors register declarations.
`assign_registers`	Lets the compiler assign variables to registers.

Table 15-29 Other optimization options

For more information on the Automatic Register Assignment option, see "Using register variables" earlier in this chapter.

With `honor_register` and `assign_registers`, you can get one of the three possible settings for the Automatic Register Assignment option, plus an extra one that wasn't included.

If honor_ register is…	And assign_ registers is…	You get this setting…
On	Off	Automatic Register Assignment option off
On	On	Automatic Register Assignment option on, and Honor 'register' first option on
Off	On	Automatic Register Assignment option on, and Honor 'register' first option off
Off	Off	Put nothing in registers

Table 15-30 Automatic Register Assignment option